

Model checking lots of systems

Andreas Classen*,
Patrick Heymans,
Pierre-Yves Schobbens
PReCISE Research Centre
Faculty of Computer Science
University of Namur
Namur, Belgium
{acs,phe,pys}
@info.fundp.ac.be

Axel Legay
IRISA/INRIA Rennes
Rennes, France
axel.legay@irisa.fr

Jean-François Raskin
Computer Science
Department
Faculty of Sciences
Université Libre de Bruxelles
(U.L.B.)
Bruxelles, Belgium
jraskin@ulb.ac.be

ABSTRACT

In software product line engineering, systems are developed in *families* and differences between family members are expressed in terms of *features*. Formal modelling and verification is an important issue in this context as more and more critical systems are developed this way. Since the number of systems in a family can be exponential in the number of features, two major challenges are (a) scalable modelling and (b) efficient verification of system behaviour. Currently, few approaches attempt to address these challenges in the context of software product lines. Those who do fail to recognise the importance of features as a unit of difference, and do not offer means for automated verification.

In this paper, we tackle those challenges at a fundamental level. Our first contribution is a feature-aware extension of transition systems, a formalism designed to describe the combined behaviour of an entire system family. The second contribution is a tool-supported model checking technique which allows to verify LTL properties against such transition systems. An empirical evaluation shows substantial gains over classical approaches.

1. INTRODUCTION

A software product line (SPL) is traditionally defined as “a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way” [11]. Software product line engineering (SPLE) thus promotes reuse throughout the software lifecycle in order to benefit from economies of scale when developing lots of similar systems. SPLE has proven beneficial to the development of embedded systems, which makes formal modelling and verification in

*FNRS Research Fellow

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2009 ACM X-XXXXX-XX-X/XX/XX ...\$10.00.

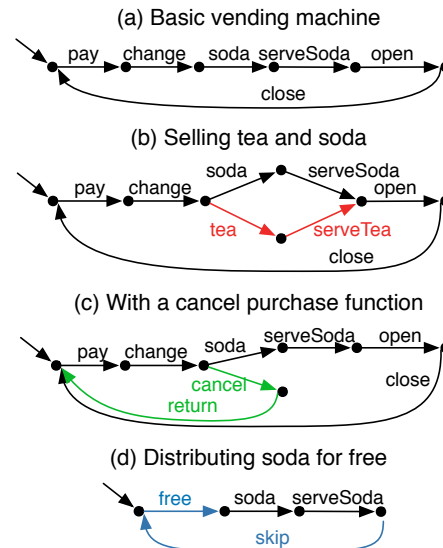


Figure 1: Several variants of a vending machine.

SPLE all the more important as many of these embedded systems are (safety) critical [14].

The differences between the systems of an SPL (i.e. its *variability*) are usually expressed using *features*, first-class abstractions that shape the reasoning of the engineers and other stakeholders [9]. A set of features can be seen as the specification of a *product*, i.e. a particular *member* of the product line. Feature diagrams (FDs) [20, 25] are commonly used to model the variability of the SPL. An FD such as the one in Figure 2 expresses the set of valid products, and since products are combinations of features, there might be an exponential number of them. Because of this, it is unreasonable to specify or verify the behaviour of each product separately.

1.1 Motivating example

Consider the example of a beverage vending machine (inspired from [16]), the running example used throughout this paper. In its basic version, the vending machine takes a coin, returns change, serves soda, opens a compartment so that the customer can take her soda, and then closes it again. This is modelled by the transition system (TS) shown in

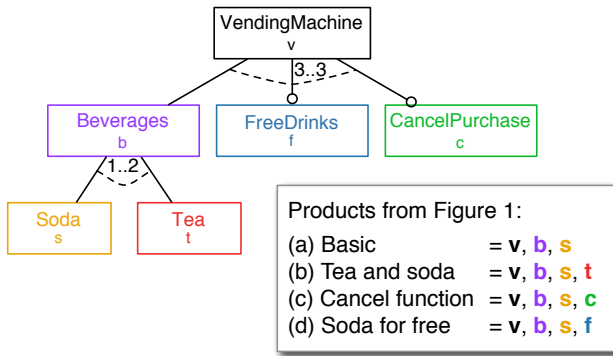


Figure 2: FD for the vending machines of Figure 1.

Figure 1(a). Adding to this basic version, there are several other variants, as for instance a machine that also sells tea, shown in Figure 1(b). A second variant lets the buyer cancel her purchase after entering a coin, Figure 1(c). A third is a machine that offers free drinks (and has no closing beverage compartment), Figure 1(d).

By combining these variants, yet other vending machines can be obtained. In fact, these four products are part of a larger SPL, which in terms of *features* is modelled by the FD of Figure 2. Basically, this FD formally describes the set of vending machine variants, in this case there are twelve of them. This means that a model of the behaviour of a small example such as this would already require twelve, mostly identical, behavioural descriptions, four of which are shown in Figure 1. For realistic cases, this number is so high that it is outright impossible to verify, let alone model, each product individually.

1.2 Current challenges

There are thus two challenges that model-based SPLE approaches need to address: (a) scaleable modelling and (b) efficient verification of system behaviour. Currently, few approaches attempt to address these challenges in the context of SPLE. Those who do, fail to recognise the importance of features as a unit of difference, and most approaches focus solely on modelling. Models and formalisms currently proposed include UML with a profile for variability [29, 28], modal I/O automata [22], modal transition systems [17, 16], deontic logics to characterise behaviours [3] and CSS extended with a product line choice operator [19].

Current proposals suffer from two limitations, both of which are addressed in the present paper. Firstly, their behavioural models generally ignore the underlying features of the family. This means that they capture different behaviours, but offer little to no means to trace products to and from their behavioural descriptions. It also means that they cannot make use of information expressed in a variability model, such as incompatibility of features. Secondly none of the approaches provide implemented methods for verifying their models against temporal properties.

1.3 Contribution

In this paper, we tackle those challenges at a fundamental level. Our first contribution are *featured transition systems* (FTS), a variant of transition systems designed to describe the combined behaviour of an entire system family. FTS has a parameterised semantics, allowing to obtain the be-

haviour of each product of the SPL. The second contribution is a proof-of-concept model checking tool as part of a model checking approach. The tool allows to verify LTL properties for all the products of an SPL at once and pinpoints the products that violate (resp. satisfy) the properties. We applied the tool to a specification exemplar in order to evaluate the approach empirically. Model checking the 64-product SPL showed a 250% speedup over the classical approach (i.e. verifying each product individually).

The principal advantages of FTS over existing work are (i) the modelling of variability as a first-class citizen, referenced from the behavioural model, (ii) the ability to reason about the whole product line, or subsets of it, (iii) the ability to model very detailed behavioural variations, (iv) the provided model checking tools, and (v) the ability to take dependency and incompatibility relations between features into account.

The paper is structured as follows. Section 2 provides the necessary background on FDs and TS. FTS are introduced in Section 3, and the model checking approach in Section 4. This is followed by an evaluation on a specification exemplar, the mine pump controller [21] in Section 5, and a discussion of further FTS extensions in Section 6. Section 7 surveys related work and Section 8 concludes the paper.

2. BASE CONCEPTS

In this section, we recap basic concepts and definitions that will be used throughout the rest of the paper. We assume that the reader is familiar with automata theory and has some basic knowledge of formal verification and model checking (otherwise, see [7, 5]).

We informally recall the definition of feature diagrams (see [25] for details) (FDs in short). Roughly speaking, an FD is a tuple (N, r, DE) where N is a set of features, $r \in N$ is the root, and $DE \subseteq N \times N$ is the set of decomposition edges between features. The semantics of an FD d , noted $\llbracket d \rrbracket_{FD}$, is the set of valid products, i.e. a set of sets of features: $\llbracket d \rrbracket_{FD} \subseteq \mathcal{P}(N)$. As an example, the semantics of the vending machine FD from Figure 2 is as follows (using the short feature names):

$$\begin{aligned} & \{ \{v, b, t\}, \{v, b, t, f\}, \{v, b, t, c\}, \{v, b, t, f, c\}, \{v, b, s\}, \\ & \{v, b, s, f\}, \{v, b, s, c\}, \{v, b, s, f, c\}, \{v, b, s, t\}, \\ & \{v, b, s, t, f\}, \{v, b, s, t, c\}, \{v, b, s, t, f, c\} \}. \end{aligned}$$

In this paper, behaviour of individuals will be represented with transition systems [2, 7] (TSs in short). A TS is a directed graph whose transitions are labelled with actions and states are labelled with atomic propositions. Formally, we have the following definition.

DEFINITION 1 (TRANSITION SYSTEM). *A TS is a tuple $M = (S, Act, trans, I, AP, L)$ where*

- S is a set of states,
- Act is a set of actions,
- $trans \subseteq S \times Act \times S$; is a set of transitions, with $(s_1, \alpha, s_2) \in trans$ sometimes noted $s_1 \xrightarrow{\alpha} s_2$,
- $I \subseteq S$ is a set of initial states,
- AP is a set of atomic propositions,
- $L : S \rightarrow 2^{AP}$ is a labelling function,

An *execution* (also called behaviour) of M is an infinite sequence $s_0\alpha_1s_1\alpha_2\dots\alpha_n s_n$ with $n \geq 0$ and $s_0 \in I$ such that $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all $0 \leq i < n$. The semantics of a TS, denoted $\llbracket t \rrbracket_{TS}$, is given by its set of executions.

In the paper, we mainly focus on two types of properties: (i) *reachability* properties, such as mutual exclusion or safety, and (ii) *temporal properties*, such as those expressible in LTL. In this paper, we will follow the classical automata-based approach [27], where those properties are represented with finite-state automata. More precisely, reachability properties will be represented by finite-word automata (FA in short), while temporal properties will be represented with Büchi automata (BA in short). A system satisfies a temporal (resp. reachability) property if (any prefix of) all of its executions are included in the language of the BA (resp. FA) that represents the property [7, 5]. We conclude the section with a formal definition for FA and BA.

DEFINITION 2. An FA (resp. BA) is a tuple $(Q, \Sigma, \delta, Q_0, F)$ where Q is a set of states, Σ is the alphabet, $\delta \subseteq Q \times \Sigma \times Q$ the transition relation, $Q_0 \subseteq Q$ a set of initial states and $F \subseteq Q$ a set of accepting states.

3. FEATURED TRANSITION SYSTEMS

In order to concisely model the behaviour of each product in the SPL, we follow existing approaches [12, 17, 19] which consist in creating a single parameterised model that can be instantiated differently for each product of the SPL. In contrast to existing approaches, however, we explicitly model which behaviour is caused by which feature, at the level of individual transitions.

3.1 Syntax

The syntax of FTS accounts for the fact that the addition of a feature to a system modifies the behaviour of this system. Consider the vending machine example. Figures 1(b,c,d) show the impact of adding features Tea, CancelPurchase and FreeDrinks, respectively, to a machine serving only soda. Tea adds two transitions (tea and serveTea); CancelPurchase also adds two transitions; and FreeDrinks replaces transitions pay and change by a single transition free as well as open/close by skip.

In order to concisely model the effects of several features on a system, our approach consists in modelling a system that contains all features, as well as annotations that indicate which transitions of the model correspond to which feature. In order to be able to express cases in which a feature *removes*, rather than adds, transitions we use a priority relation over alternative transitions.¹ This leads us to define a *featured* TS as a TS in which each transition is labelled with a feature, and where a priority relation may be associated to transitions leaving the same state.

The FTS for the vending machine example is shown in Figure 3. The feature label of a transition is shown next to its action label, separated by a slash (in addition, the transitions are coloured in the same way as the features in Figure 2). Intuitively, the FTS captures the impact of all features in a single diagram. An FTS is thus a TS in which transitions are labelled with features and a priority relation. Formally, this is defined as follows.

¹In essence, removing a transition corresponds to adding an alternative transition of higher priority.

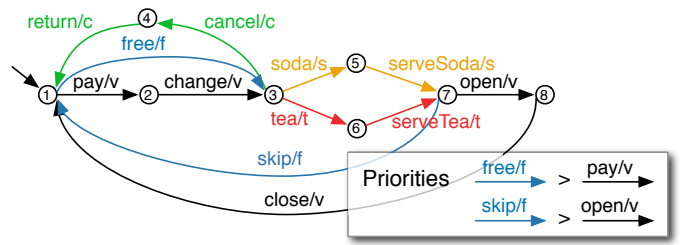


Figure 3: FTS of the vending machine.

DEFINITION 3 (ABSTRACT SYNTAX OF FTS). An FTS dt is a tuple $dt = (S, Act, trans, I, AP, L, d, \gamma, >)$ where

- $(S, Act, trans, I, AP, L)$ is a TS,
- $d = (N, r, DE)$ is an FD,
- $\gamma : trans \rightarrow N$ is a total function, labelling transitions with features,
- $> \subseteq trans \times trans$ is a partial order, defining priorities between transitions.

Transition priorities offer an intuitive way to model cases in which one feature overrides the behaviour of another one. If a transition t labelled with feature f has priority over transition t' labelled with feature f' , this means that products containing both f and f' only have transition t (they would have both transitions if there were no priority relation). The transition free of the FreeDrinks feature, for instance, has priority over pay (which belongs to the root feature, VendingMachine). The result is that pay will not appear in any product that contains the feature FreeDrinks, such as the one in Figure 1(d).²

A common modelling pattern is that the behaviour of a child feature (wrt. the FD) overrides the behaviour of its parents. Formally, we can use the decomposition relation DE of the FD (a directed graph), to induce the priority relation of the FTS.

DEFINITION 4 (PRIORITIES INDUCED BY FD). A transition $s \xrightarrow{\alpha} s_1$ labeled with f_1 has priority over $s \xrightarrow{\beta} s_2$ labeled with f_2 , $s \xrightarrow{\alpha} s_1 > s \xrightarrow{\beta} s_2$, iff f_2 is an ancestor of f_1 , i.e. if the transitive closure of DE has an edge from f_2 to f_1 .

The purpose of an FTS is to model the behaviour of the whole SPL. From the FTS, one can obtain the behaviour of one particular product through *projection*. Intuitively, the diagrams (a), (b) and (c) of Figure 1 can be obtained by removing the transitions of all but one feature (or colour) from Figure 3 (keeping the black transitions). In Figure 1(d), because of the priorities discussed above, the transitions pay and open are removed as well.

Formally, in order to obtain the behaviour of a particular product, one *projects* the FTS of the SPL on the corresponding set of features, say $p \in \llbracket d \rrbracket_{FD}$. This transformation is entirely syntactical and consists in removing (i) all transitions linked to features that are not in p , and (ii) all transitions that are supplanted by higher priority transitions. The result of the projection is an ordinary TS.

²Since state 2 and transition change become unreachable, they are omitted from the diagram.

DEFINITION 5 (PROJECTION). *The projection of an FTS dt to a product $p \in \llbracket d \rrbracket_{FD}$, noted $dt|_p$, is the TS $t = (S, Act, trans', I, AP, L)$ where*

$$trans' = \left\{ (s_1, \alpha, s_2) \mid (s_1, \alpha, s_2) \in trans \wedge \gamma(s_1, \alpha, s_2) \in p \right. \\ \wedge \nexists (s_1, \alpha', s_2') \in trans \bullet \gamma(s_1, \alpha', s_2') \in p \\ \left. \wedge (s_1, \alpha', s_2') > (s_1, \alpha, s_2) \right\}.$$

Note that the concept of *parallel composition* also exists for FTS. If the SPL to be modelled consists of several processes running in parallel, each process can be modelled as a separate FTS, all sharing the same underlying features and FD. The FTS of the system can then be obtained by composing these processes. For FTS, one can easily adapt the well-accepted handshake communication model, whereby the execution of parallel processes is synchronised on transitions with shared actions and otherwise interleaved.

3.2 Semantics

Each TS obtained through projection describes the behaviour, as in Definition 1, of a particular product of the SPL. The semantics of an FTS is thus the union of the behaviours of the projections on all valid products.

DEFINITION 6 (SEMANTICS OF AN FTS).

$$\llbracket dt \rrbracket_{FTS} = \bigcup_{c \in \llbracket d \rrbracket_{FD}} \llbracket dt|_c \rrbracket_{TS}$$

An important observation is that, except for trivial cases, the semantics of an FTS we just defined is not equal to the usual TS semantics (as given by Definition 1). Formally, there exists an FTS dt for which $\llbracket dt \rrbracket_{FTS} \neq \llbracket TS(dt) \rrbracket_{TS}$, where $TS(dt)$ is the TS obtained by removing d, γ and $>$ from dt . The vending machine SPL is an example for such an FTS. Indeed, an execution e in which the vending machine would ask the first customer for a coin and offer a free drink to the next one would be part of $\llbracket TS(dt) \rrbracket_{TS}$, since in a TS, the choice between **pay** and **free** is non-deterministic. However, the execution does not correspond to any of the machines in the SPL: these should either always offer free drinks or always require payment, and hence $e \notin \llbracket dt \rrbracket_{FTS}$. More generally, we have the following.

THEOREM 7 (FTS SEMANTICS VS. TS SEMANTICS).

$$\forall dt \bullet \llbracket dt \rrbracket_{FTS} \subseteq \llbracket TS(dt) \rrbracket_{TS}$$

This theorem illustrates that one cannot simply use classical model checking algorithms to verify properties for the complete SPL. Indeed, while this verification might be sound, it is not always complete (i.e. such an approach would find false positives). Definition 6 shows another problem that we have to face when model checking an FTS: the exponential blowup caused by considering all products of the SPL. This adds to the state-explosion problem that already exists in classical model checking.

These considerations justify the need for an FTS-specific model checking algorithm, but before we get there, we need a more operational definition of the FTS semantics.

3.3 Reachability in FTS

Model checking algorithms perform a search in the state space of the FTS and thus need an execution model that is

faithful to the FTS semantics. As we just showed, TS semantics does not apply to FTS, mainly because it ignores priorities attached to transitions. In addition, we want our model checking approach to report for which products a property does (or does not) hold. To explore the state space of an FTS, we thus need an execution model that keeps track of products and respects transition priorities.

In consequence, we define the reachability relation R_0 for an FTS, which is constructed as the state space is explored, as a set of couples $R_0 \subseteq S \times \mathcal{P}(\mathcal{P}(N))$ where $(s, px) \in R_0$ means that state s is reachable by the products in px . In particular, the initial states of the FTS are reachable for all products, defined formally as follows.

DEFINITION 8. *Initially reachable states of an FTS are*

$$I_0 \triangleq \{(s, \llbracket d \rrbracket_{FD}) \mid s \in I\}.$$

Given a state s reachable by products in px , a transition leaving s , $s \xrightarrow{\alpha} s'$, can be fired for all products if its feature is part of all products in px and if there is no higher-priority transition $s \xrightarrow{\alpha'} s''$ which could also be fired. If these requirements are met, s' is reachable by the same products in px . In case the requirements are not met, the transition cannot be fired for all products, meaning that s' will only be reachable by a subset of px :

- If there is a higher-priority transition that could equally be fired, then the transition can only be fired for (and thus s' is only reachable by) the products which do not contain the feature of the alternative transition.
- If the feature that labels the transition is part of some products in px only, then it can only be fired for (and thus s' is only reachable by) these products.
- If none of the products contains the feature that labels the transition, it cannot be fired at all.

This is formalised in the definition below.

DEFINITION 9. *The successors of a state $s \in S$ reachable by products in $px \in \mathcal{P}(\mathcal{P}(N))$ can be computed with the following operator*

$$Post_0(s, px) \triangleq \left\{ (s', px') \mid s \xrightarrow{\alpha} s' \in trans \wedge \right. \\ \left. px' = \{p \in px \mid \gamma(s \xrightarrow{\alpha} s') \in p \wedge \right. \\ \left. \{\gamma(s \xrightarrow{\alpha'} s'') \mid s \xrightarrow{\alpha'} s'' > s \xrightarrow{\alpha} s'\} \cap p = \emptyset \} \right\}.$$

Let us illustrate this with the vending machine FTS in Figure 3. Indeed, state 1 is an initial state, and thus reachable by all products. From there, the transition **pay** can only be fired by products containing the feature **v** (the label of **pay**), and not containing the feature **f** (the label of the higher-priority transition **free**). And thus, state 2 is reachable by these products only. From state 2, transition **change** can be fired for all products containing **v**, and so state 3 is reachable by the same products as state 2.

Recording the set of products would be too expensive. In the worst case, the set of reachable states will be of size $|S| \cdot 2^{2^{|N|}}$. We thus propose a more concise representation for a set of products: by stating which features the products must have (required features) and which they cannot have (excluded features). This symbolic data structure is defined as follows.

DEFINITION 10. A triplet $(s, rf, ef) \in S \times \mathcal{P}(N) \times \mathcal{P}(N)$ is a symbolic representation for a tuple $(s, px) \in S \times \mathcal{P}(\mathcal{P}(N))$ such that $\llbracket (s, rf, ef) \rrbracket \triangleq (s, \{p \in \llbracket d \rrbracket_{FD} \mid rf \subseteq p \wedge ef \cap p = \emptyset\})$.

The efficient reachability relation R is thus a set of triplets (s, rf, ef) , where the initially reachable states I and the successors $Post$ are defined as follows.

DEFINITION 11. In symbolic representation, the initially reachable states of an FTS are $I \triangleq \{(s, \emptyset, \emptyset) \mid s \in I\}$, and the successors of a state $s \in S$ reachable by products containing features $rf \subseteq N$ and not containing features $ef \subseteq N$ are

$$\begin{aligned} Post(s, rf, ef) \triangleq & \{(s', rf', ef') \mid s \xrightarrow{\alpha} s' \\ & \wedge rf' = rf \cup \{\gamma(s \xrightarrow{\alpha} s')\} \\ & \wedge ef' = ef \cup \bigcup_{s \xrightarrow{\alpha} s'' > s \xrightarrow{\alpha} s'} \gamma(s \xrightarrow{\alpha} s'') \\ & \wedge ef' \cap rf' = \emptyset\}. \end{aligned}$$

It can easily be shown that the symbolic successor function is equivalent to its explicit counterpart, that is:

THEOREM 12. For any (s, rf, ef) , it holds that

$$Post_0(\llbracket (s, rf, ef) \rrbracket) = \llbracket Post(s, rf, ef) \rrbracket$$

Where $\llbracket \cdot \rrbracket$ is trivially extended to sets of triplets.

With this, the reachable states are the initially reachable states and those that can be reached from them, which can be expressed by the least fixpoint of the successor relation.

DEFINITION 13. The symbolic reachability relation $R \subseteq S \times \mathcal{P}(N) \times \mathcal{P}(N)$ for an FTS is defined as

$$R = \mu X \bullet I \cup Post(X),$$

where $Post$ is extended to sets of triplets as follows: $Post(x) = \bigcup_{(s, rf, ef) \in x} Post(s, rf, ef)$. The relation can be calculated following Tarski's fixpoint theorem by applying the successor relation until it stabilises, i.e.

$$R = I \cup Post(I) \cup Post(Post(I)) \cup \dots = I \cup \bigcup_{i \geq 1} Post^i(I).$$

Using this representation, the size of the reachability relation will be $|S| \cdot 2^{|N|} \cdot 2^{|N|}$ in the worst case, which is significantly smaller than what we had previously. In addition, its size can be further reduced by exploiting the following property: if a state s is known to be reachable by products in px , then it is also reachable by the products in any subset of px . Formally, if $(s, px) \in R_0$ and $(s, px') \in R_0$ with $px' \subseteq px$, it is sufficient to only keep (s, px) in R_0 . In general, it is sufficient to keep the maximal elements of the partial order induced by the subset relation \subseteq over the $\{px \mid (s, px) \in R_0\}$ for each state s . In terms of the symbolic representation we are using, an equivalent partial order can be defined as follows.

DEFINITION 14. For a state $s \in S$ and a set $R \subseteq s \times \mathcal{P}(N) \times \mathcal{P}(N)$, the relation \sqsubseteq , a partial order over R , is defined as $(s, rf, ef) \sqsubseteq (s, rf', ef') \triangleq rf \supseteq rf' \wedge ef \supseteq ef'$.

In the same vein, testing whether a state s is reachable by products in px could be done by checking whether (s, px) is in the reachability relation R_0 , but such a check is too fine-grained. Indeed, it is sufficient if there is a $(s, px') \in R_0$ such that $px \subseteq px'$. In the symbolic representation this translates to checking whether for some (s, rf, ef) there is a $(s, rf', ef') \in R$ such that $(s, rf, ef) \sqsubseteq (s, rf', ef')$.

4. MODEL CHECKING FTS

As stated in Section 2, we follow the well established approach of automata-based model checking [27], where reachability and temporal properties are expressed by automata. The first step towards model checking FTS is thus to define the composition of an automaton and an FTS, commonly called *synchronous product*.

4.1 Synchronous product

Automata-based model checking algorithms check whether the automaton obtained by taking the synchronous product of the system with the automaton representing the negation of the property is empty or not. Intuitively, the synchronous product will remove from the system all the (prefixes of) executions that are not member of the language of the automaton representing the property. For FTS, it is formally defined as follows.

DEFINITION 15 (SYNCHRONOUS PRODUCT). For an FTS $dt = (S, Act, trans, I, AP, L, d, \gamma, >)$ and an FA/BA $a = (Q, \mathcal{P}(AP), \delta, Q_0, F)$, the synchronous product is an FTS $dt \otimes a = (S \times Q, Act, trans', I', AP', L', d, \gamma', >')$, where

- $AP' = Q$ and $L'(s, q) = q$, i.e. the new FTS is labeled with the states of the FA/BA
- $(s, q) \xrightarrow{\alpha'} (t, p)$ iff $s \xrightarrow{\alpha} t \wedge q \xrightarrow{L(t)} p$
- $\gamma'((s, q) \xrightarrow{\alpha'} (t, p)) = \gamma(s \xrightarrow{\alpha} t)$
- $(s, q) \xrightarrow{\alpha'} (t, p) >' (s', q') \xrightarrow{\alpha'} (t', p')$ iff $s \xrightarrow{\alpha} t > s' \xrightarrow{\alpha'} t'$
- $I' = \{(s_0, q) \mid s_0 \in I \wedge \exists q_0 \in Q_0 \bullet (q_0, L(s_0), q) \in \delta\}$, i.e. the initial states are those that can be reached from an initial state of the FA/BA.

4.2 FTS model checking scenarios

Our objective is to verify temporal/reachability properties on FTSs in such a way that (a) if the property is satisfied by the FTS, then it is also satisfied by every product of the SPL, and (b) if the property is violated, the algorithm reports a counter example as well as details on which products of the SPL violate the property. This differs from classical model checking algorithms for TSs that in the case of a violation just return the counter example showing why the property is not satisfied. The additional information is intended to help the analyst when correcting the model. Formally, satisfaction is defined as follows.

DEFINITION 16 (SATISFACTION IN FTS). An FTS dt satisfies a (temporal, or reachability) property ϕ , iff

$$\forall p \in \llbracket d \rrbracket_{FD} \bullet dt|_p \models \phi.$$

Extending the \models relation, we note this $dt \models \phi$.

The FTS model checking problem can now be formalised.

DEFINITION 17 (MC(FTS, ϕ)). Given a property ϕ and an FTS dt , $MC(dt, \phi)$ returns true iff $dt \models \phi$. If $dt \not\models \phi$, it returns false, a counter example e , and a non-empty set of products $px \subseteq \llbracket d \rrbracket_{FD}$ such that $\forall p \in px \bullet dt|_p \not\models \phi$ with e as counter example.

The basic model checking scenario is analogous to classical model checking: just as the returned counter example might be one among several violating traces, the set of violating products is not necessarily complete. In case of a violation, it is therefore not possible to know whether there are products that *do* satisfy the property. This gives rise to a model checking problem specific to SPLE: determine which products satisfy and which violate the property.

DEFINITION 18 (EXTMC(FTS, ϕ)). *Given a property ϕ and an FTS dt , ExtMC(dt, ϕ) returns true iff $dt \models \phi$. If $dt \not\models \phi$, it returns false and a set c of couples (e, px) where e is a counter example and px a non-empty set of products such that $\forall p \in px \bullet dt|_p \not\models \phi$. Furthermore, it holds that*

$$\forall p \in \llbracket d \rrbracket_{FD} \bullet p \notin \bigcup_{(e, px) \in c} px \implies dt|_p \models \phi.$$

The additional requirement states that the list of counter examples has to be exhaustive, i.e. all products that are not mentioned satisfy the property. The procedure thus implicitly returns a set of violating and a set of satisfying products. A further variation of these two scenarios is useful for SPLE: limiting the verification to a subset of the products of the SPL. Basically, both scenarios would take as an additional parameter the set of products $px \subseteq \llbracket d \rrbracket_{FD}$ to verify. From there on, the definitions are analogous.

4.3 Reachability (and safety) checking

While the reachability relation can be obtained by computing the fixpoint expression given in Definition 13, it is generally more efficient to verify properties directly with a depth-first search (DFS) in the graph.

Such a DFS is implemented in the procedure to the right, **CheckInvariant**. The procedure is parameterised so that it can be used for all model checking scenarios identified in the previous section. It takes four parameters: (i) the FTS, (ii) the invariant property to be checked, (iii) a boolean flag instructing the algorithm to stop upon discovering a violation, and (iv) the set of products to be verified (using the symbolic representation from Definition 10).

The algorithm basically computes the symbolic reachability relation defined in Section 3.3. It maintains the set of reachable states R , a stack T of triplets (s, rf, ef) and a set of property violations bad . The initial states are always reachable for all products that are going to be verified, and R is initialised accordingly (line 1). The algorithm iterates over the initial states (line 4) and performs a DFS for each one (line 8). It first checks whether the current state is bad (line 10); if it is, the current trace (that is, the counter example) and the products for which the bad state is reachable (rf, ef) are saved (line 11). If the *break* flag is set, the algorithm will terminate here (line 12).

The procedure continues by calculating the set of unvisited successors of the current state (line 14). For this, it calls the sub procedure **NewPost** shown on the right, which uses the *Post* operator from Definition 11 to determine the successors, and filters out those that are already in R (with the optimisation discussed in Section 3.3). If all successors were visited, the algorithm backtracks (line 16). Otherwise the search proceeds with one of the successor states, which is added to R (line 19) again using the optimisation detailed in Section 3.3 (intuitively, max_{\sqsubseteq} removes the redundant triplets from R).

Input: An FTS $dt = (S, Act, trans, I, AP, L, d, \gamma, >)$, an invariant property ϕ , a boolean flag *break* meaning to stop upon discovering a violation of ϕ , a set of required (resp. excluded) features rf_0 and ef_0 to delimit the products to explore.

Output: *True* if all states satisfy ϕ , otherwise *false* and a set of quadruplets (state, set of required, set of excluded features, error trace) with the violations.

```

1  $R \leftarrow \{(s_0, rf_0, ef_0) \mid s_0 \in I\}$ ;           % reachable states
2  $Trace \leftarrow []$ ;                               % current trace
3  $bad \leftarrow \emptyset$ ;                            % set of bad states
4 while  $I \neq \emptyset$  do
5   Take  $s_0$  from  $I$ ;
6    $I \leftarrow I \setminus \{s_0\}$ ;
7    $push((s_0, rf_0, ef_0), Trace)$ ;
8   while  $Trace \neq []$  do
9      $(s, rf, ef) \leftarrow top(Trace)$ ;
10    if  $s \not\models \phi$  then
11       $bad \leftarrow bad \cup \{(s, rf, ef, Trace)\}$ ;
12      if break then return false, bad
13    end
14     $unvisited \leftarrow NewPost(dt, R, s, rf, ef)$ ;
15    if  $unvisited = \emptyset$  then
16       $pop(Trace)$ ;
17    else
18      Take  $(s', rf', ef') \in unvisited$ ;
19       $R \leftarrow max_{\sqsubseteq}(R \cup \{(s', rf', ef')\})$ ;
20       $push((s', rf', ef'), Trace)$ 
21    end
22  end
23 end
24 return  $(bad = \emptyset), bad$ 

```

Procedure CheckInvariant($dt, \phi, break, rf_0, ef_0$)

Note that in its current form, the algorithm does not take the structural information of the FD into account. This is important insofar that a symbolic couple (rf, ef) might not designate a product of the FD at all, while still being considered during the DFS. Concretely, in such a case, the DFS would visit states that are not actually reachable. While this overhead could be deemed acceptable, it is a problem if the procedure finds a bad state which is not in fact a bad state because it belongs to no valid product. In order to solve the problem, one could easily add a validity check inside **NewPost**, so that it only returns valid unvisited states, i.e. $\{(s, rf, ef) \mid \exists p \in \llbracket d \rrbracket_{FD} \bullet rf \subseteq p \wedge ef \cap p = \emptyset\}$. This

Input: An FTS dt , a reachability relation R , a state s , a set of required rf and a set of excluded features ef .

Output: The successor states of (s, rf, ef) , i.e. $Post()$, except for those that are already reachable.

return $\left\{ \begin{array}{l} (s', rf', ef') \in Post(s, rf, ef) \\ \exists (s', rf'', ef'') \in R \\ \bullet (s', rf', ef') \sqsubseteq (s', rf'', ef'') \end{array} \right\}$;

Procedure NewPost(dt, R, s, rf, ef)

check can be implemented efficiently with a SAT call [23].

Let ϕ be a reachability property, $FA(\neg\phi)$ the negation of the property transformed into an FA, and $F_{FA(\neg\phi)}$ the set of accepting states of that FA, the model checking scenario $MC(dt, \phi)$ amounts to computing

$$\text{CheckInvariant}(dt \otimes FA(\neg\phi), \bigwedge_{s \in F_{FA(\neg\phi)}} \neg s, true, \emptyset, \emptyset)$$

and expanding the symbolic representation of the products returned in case of violation. The $ExtMC(dt, \phi)$ scenario is almost identical, with the break flag set to *false*.

4.4 Temporal property checking

Checking arbitrary temporal properties, e.g. specified in LTL, requires only a small addition to the **CheckInvariant** procedure. In addition to checking the reachability of a bad state, the algorithm checks whether a bad state is also on a cycle. This is implemented with a double DFS in procedure **CheckPersistence**: once the outer DFS finds a reachable bad state, the inner checks whether it is reachable from itself. Both DFS are almost identical to **CheckInvariant**, and are not detailed any further.

Let ϕ be a temporal property, $BA(\neg\phi)$ the negation of the property transformed into a BA, and $F_{BA(\neg\phi)}$ the set of accepting states of that BA, the model checking scenario $MC(dt, \phi)$ for a temporal property amounts to computing

$$\text{CheckPersistence}(dt \otimes BA(\neg\phi), \bigwedge_{s \in F_{BA(\neg\phi)}} \neg s, true, \emptyset, \emptyset)$$

and expanding the symbolic representation of the products. Again, $ExtMC(dt, \phi)$ is obtained with *break* set to *false*.

5. EVALUATION

5.1 Theoretical evaluation

The bottleneck of any automata-based model checking procedure is the construction of the BA that accepts the negation of the property ϕ to check. In the worst case, the BA is exponential in the size of the formula ϕ . A natural upper bound for LTL model checking is therefore $O(|TS|.2^{|\phi|})$. The lower bound for this problem is PSPACE-Hard [24].

An upper bound for FTS LTL model checking of a property ϕ on an FTS with n features is $O(|FTS|.2^{|\phi|+n})$, since in the worst case the algorithm in procedure **CheckPersistence** will have to verify all possible feature combinations. As of the lower bound, the FTS LTL model checking problem is also at least PSPACE-Hard, since one can easily reduce the standard LTL model checking problem to model checking LTL in FTS. There also exists a linear reduction the other way round, which means that the problem is indeed PSPACE-Hard.

5.2 Empirical evaluation

We implemented the FTS model checking approach with the functional programming language Haskell [8].³ The tool comes in the form of a library that can be loaded into a Haskell interpreter, resulting in a command line interface, or compiled to execute verifications in batch mode. The advantage of using Haskell as implementation language is its pervasive use of lazy evaluation and the easy translation of

Input: An FTS $dt = (S, Act, trans, I, AP, L, d, \gamma, >)$, a persistence property ϕ , a boolean flag *break* meaning to stop upon discovering a violation of ϕ , a set of required (resp. excluded) features rf_0 and ef_0 to delimit the products to explore.

Output: *True* if ϕ is persistent, otherwise *false* and a set of quadruplets (state, set of required, set of excluded features, error trace) with the violations.

```

1  R ← {(s0, rf0, ef0) | s0 ∈ I};           % reachable states
2  Trace ← [];                               % current trace
3  bad ← ∅;                                  % set of violations
4  IR ← ∅;                                   % reachable states of inner DFS
5  ITrace ← [];                              % current trace of inner DFS
6  while I ≠ ∅ do
7    Take s0 from I;
8    I ← I \ {s0};
9    push((s0, rf0, ef0), Trace);
10 while Trace ≠ [] do
11   (s, rf, ef) ← top(Trace);
12   unvisited ← NewPost(dt, R, s, rf, ef);
13   if unvisited = ∅ then
14     pop(Trace);
15     if s ∉ ϕ then
16       % Check for cycle
17       IR ← max⊆(IR ∪ {(s, rf, ef)});
18       push((s, rf, ef), ITrace);
19       while ITrace ≠ [] do
20         (s', rf', ef') ← top(ITrace);
21         if ∃(s, rf'', ef'') ∈ Post(s', rf', ef')
22         then % Back transition found
23           bad ← bad ∪ {(s, rf, ef, Trace.ITrace)};
24           if break then return false, bad
25         else
26           unvisited' ← NewPost(dt, IR, s', rf', ef');
27           if unvisited' = ∅ then
28             pop(Trace);
29           else
30             Take
31             (s'', rf'', ef'') ∈ unvisited';
32             IR ← max⊆(IR ∪ {(s'', rf'', ef'')});
33             push((s'', rf'', ef''), ITrace)
34           end
35         end
36       end
37     end
38   else
39     Take (s', rf', ef') ∈ unvisited;
40     R ← max⊆(R ∪ {(s', rf', ef'')});
41     push((s', rf', ef'), Trace)
42   end
43 end
44 return (bad = ∅), bad

```

Procedure CheckPersistence($dt, \phi, break, rf_0, ef_0$)

³Download at www.info.fundp.ac.be/~acs/FTSHaskell

Table 1: Benchmark results for exhaustive counter example search $ExtMC(FTS, \phi)$.

Formula ϕ	Four features, four products			Nine features, 64 products		
	Cur.	Our	Diff.	Cur.	Our	Diff.
(1.1) $\Box\Diamond(start \wedge \bigcirc msg \wedge (methane \Rightarrow palarm))$ $\Rightarrow (\Box\Diamond(methane \Rightarrow \Diamond pumpoff))$	✓	9.389 s	5.563 s 68.78 %	57.706 s	8.162 s	607.04 %
(1.2) $\neg\Box\Diamond(start \wedge \bigcirc msg \wedge (methane \Rightarrow \Diamond palarm))$	✗	25.741 s	37.663 s -31.65 %	138.970 s	102.716 s	35.29 %
(1.3) $\Box\Diamond(start \wedge \bigcirc msg) \Rightarrow \Box(pumpon \Rightarrow \Diamond running)$	✓	5.084 s	4.308 s 18.01 %	13.716 s	5.317 s	157.96 %
(1.4) $\Box\Diamond(msg \wedge \bigcirc level) \Rightarrow \Box\Diamond(lowwater \Rightarrow \Diamond pumpoff)$	✓	4.970 s	4.156 s 19.59 %	16.450 s	4.926 s	233.92 %
(1.5) $\Box\Diamond(msg \wedge \bigcirc level \wedge ready)$ $\Rightarrow \Box((highwater \wedge \Box!methane) \Rightarrow \Diamond pumpon)$	✓	5.172 s	4.462 s 15.90 %	14.981 s	5.033 s	197.68 %
(1.6) $\Box\Diamond(msg \wedge \bigcirc level \wedge ready)$ $\Rightarrow \Box((highwater \wedge \Box!methane) \Rightarrow \Diamond pumpon)$	✗	5.437 s	4.405 s 23.44 %	17.741 s	4.914 s	261.00 %

mathematical formulae into program code. The tool interfaces with `ltl2ba`,⁴ based on [18], to automate LTL to BA translation and uses `Graphviz`⁵ to render FTS graphically. It makes some minor simplifications: priorities between transitions are derived from user-defined feature priorities and it uses a less efficient algorithm for persistency checking.

In order to validate our approach, we conducted a study of examples found in related work (see [8]). We report here on an analysis of the mine pump controller exemplar [21]. The system’s purpose is to keep a mine shaft clear of water while at the same time avoiding the danger of a methane related explosion. It consists of a water pump, a sensor measuring the water height and a sensor measuring the abundance of methane in the mine. The system is supposed to activate the pump once the water level reaches a preset threshold, but only if the methane abundance is below a critical limit.

The system, as implemented in [21], is composed of a base system, `base`, and three high level features: `c`, a command interface which activates or deactivates the water regulation; `m`, a methane alarm interface; and `l`, the water regulator itself. The system and its environment are modelled by four distinct FTS: the FTS representing the program itself, a second FTS that models the system state, and three other FTS that model the state of the environment: the water level, the methane level and the state of the pump. The system FTS is the parallel composition of these FTS, it has 604 states and 1306 transitions. We introduce variability by modelling `c` and `a` as optional features. With just these high level features, the SPL has four products (counting all products explicitly, the system has 1828 states and 4612 transitions). In a second step, we introduced more variability by decomposing the high level features, resulting in an SPL with nine features and 64 products (here the explicit count is 29760 states and 69856 transitions).

We used our prototype to prove properties such as those identified in [1] for both SPLs, comparing the performance of the classical model checking algorithm (also implemented in the tool) to our method. All benchmarks were run on a MacBook Pro with a 2,4 GHz Core 2 Duo processor and 4 Gb of RAM; the library was compiled using `GHC`.⁶ Each benchmark comprises the construction of the BA, the computation of the parallel composition, of the synchronous product and of an exhaustive counterexample search. The reported runtime is the average of three executions. The results are listed

in Table 1, where ✓ (resp. ✗) means property satisfied (resp. violated), ‘Cur.’ is the runtime of the classical approach (i.e. verify each product individually), ‘Our’ is the runtime of our algorithm, and ‘Diff.’ the speedup in percent.

Property (1.1), for instance, expresses the fact that “the pump shall eventually be off when the methane level is critical.” The formal property needs a fairness assumption: $\Box\Diamond start \wedge \bigcirc msg$ forces the system to progress, preventing traces that only contain environment transitions. And $\Box\Diamond methane \Rightarrow palarm$ is the assumption that in case of critical methane, the system will eventually be notified with an alarm message. Property (1.2) is used to check that the assumption does in fact hold for some products. Indeed, it holds for products having features `base` and `m`, which means that property (1.1) also holds for these products.

These results show that even with a very low number of products, our approach achieves in average a 20% improvement over the classical algorithm. An exception are cases in which the number of counterexamples is relatively high, such as for property (1.2). However, the gains over the classical approach increase dramatically with the number of products. With 64 products, the average speedup is 250% (and 64 is still a rather low number). Furthermore, the results show that our approach, as opposed to the classical one, scales with the number of products. It has to be noted that both implementations are rather naive and do not make use of known optimisation techniques. Their sole purpose is to benchmark the speedup from using our algorithms.

6. FUTURE WORK

A natural generalisation of FTS is to allow to label each transition with a Boolean expression over the set of features. Definition 5, the projection, would have to be adapted so that a transition is part of the projected TS if the interpretation of the product satisfies its Boolean expression. This would allow for greater flexibility when modelling. For instance, one could express situations in which a transition belongs to several features. Conceptually, such an extension is rather straightforward. The reachability relation will consist of couples (s, b) where s is a state and b a boolean expression characterising the products in which s is reachable. There are no fundamental changes required in the algorithms, although an efficient implementation would have to use BDDs or a SAT solver.

This paper makes the assumption that the state-space of the SPL is given directly in form of a single (or multiple parallel) FTS. While this assumption does hold for illustrative examples or small cases, it is unreasonable in practice.

⁴www.lsv.ens-cachan.fr/~gastin/ltl2ba

⁵www.graphviz.org

⁶www.haskell.org/ghc

Indeed, the intention of this work was to lay the basis for formal verification in SPLE. Hence, we do not expect the end-user to model directly in FTS. We are currently working on ways to translate high-level modelling languages, such as statecharts or Promela into FTS.

Even if the analyst uses a high-level modelling language, it is likely that the full specification of an SPL cannot be created as a single model. We are thus also exploring merging techniques, which create an FTS of the SPL based on the TS fragments of some high-level features [10].

7. RELATED WORK

Approaches for SPLE. Before we examine each related approach in detail, we note that they can be broadly categorised along two lines. On the one hand, there are approaches that provide a modelling language *with* verification mechanisms [17, 19], as we do, and others that just provide the modelling language. Among those there are a number of formal approaches which do not provide mechanisms for the verification of temporal properties [22, 16, 15, 3, 26] as well UML-based approaches [29, 28, 12, 13] which are syntactical. This means that their family models can be used to syntactically derive the model of a specific product, but that formal verification of these models is not possible. On the other hand, one can distinguish between approaches treating *variability as a first-class citizen* [12, 13] and those expressing *variability as part of the behavioural model* [29, 28, 17, 22, 16, 19, 15, 26, 3]. In [4], Bachmann *et al.* propose orthogonal variability modelling (OVM), a modelling paradigm that consists in documenting variability as first-class citizen in a central model, from which it is traced to other non-variability models, called *base models*. We believe that OVM has a number of important benefits, the main of which is a clear separation of concerns. Approaches that represent variability as part of the behavioural model are problematic for several reasons. Firstly, they bury variability information inside a behavioural model even though variability crosscuts all kinds of models, not only behaviour. Secondly, when variability information is intermingled with other base models, the variability in these models has to be kept in sync. As the product line evolves, for instance, optional functionality might become mandatory, requiring similar changes in all base models. Thirdly, variable artefacts in these approaches do not have an identity other than what is provided by the modelling language, making it hard to explicitly capture the notion of a *product* as a set of features or decisions.

Ziadi *et al.* [29, 28] propose a UML profile for variability with stereotypes for optionality, alternatives and refinement (called ‘virtuality’). This profile can be used to model product line behaviour with UML sequence diagrams. The approach does not provide verification mechanisms nor does it use a first-class variability approach.

Czarnecki, Antkiewicz and Pietroszek [12, 13] propose a pruning-based approach to UML modelling for SPLE which follows OVM and separates variability from the base models. They propose to annotate model fragments with ‘presence conditions’, i.e. Boolean expressions over features that define to which products a fragment belongs. The authors do not deal with semantic issues and only verify syntactical correctness of possible projections.

Larsen *et al.* propose modal I/O automata as a way to model configurable components and provide a formal notion

of compatibility between components [22]. Their notion of variability is limited to that of variable component interfaces and the approach does not deal with the problem of specifying variable behaviour in order to verify temporal properties.

Modal transition systems (MTS) were first proposed by Fischbein *et al.* [17] to model SPL behaviour. Transitions in an MTS are mandatory or optional. An MTS thus specifies a family of ordinary TS obtained by removing optional transitions. Similarly to our approach, a single MTS model check allows to verify all possible products at once. However, MTS lack the notion of feature and priority between transitions. ~~Intuitively, they execute without a memory of taken decisions. They are thus less expressive than FTS. For instance, since each optional transition may or may not be included in a TS, there is no way of modelling a system in which two optional transitions occur together or not at all.~~

Fantechi and Gnesi [16, 15] extended this approach by introducing explicit variability operators into MTS, similar to the way Ziadi *et al.* did for UML. In addition to optionality, they allow to specify cases in which *i..j* outgoing transitions may be taken. This proposal does not overcome the inherent MTS limitation that individual variation points are unrelated. Asirelli *et al.* show how an MTS can be completely characterised with deontic logic formulae [3], which means that deontic logic is as expressive as MTS and could be used as a specification language as well.

Gruher *et al.* propose PL-CSS, a variant of CSS extended with a product line variant operator that allows to model an alternative choice between two processes [19]. The semantics of PL-CSS is given in terms of multi-valued modal Kripke structures, which is similar to our FTS notation. The goal of their verification procedure is similar to ours, be able to verify all systems of the product line in one shot. However, their model checking procedure is only sketched (and as far as we know) no implementation is available. Also properties have to be expressed with multi-valued modal μ -calculus, whereas we use the more commonly known LTL. Moreover, we do not introduce a new operator, making it easier to adapt existing tools for our approach. Also, modelling variability with the alternative choice operator can result in verbose descriptions since common parts of the alternatives have to be duplicated [8].

Other approaches. In addition to these SPL-specific approaches, there is a body of related research in the field of feature interaction detection [6]. The purpose of these approaches is to detect and manage incompatibilities, called *harmful interactions*, between features (mostly in telecommunication systems). However, feature interaction research lacks the ~~system modelling and the~~ product line perspective we focus on.

In the context of workflow modelling, van der Aalst *et al.* propose configurable workflows, that is, workflow templates that contain variation points [26]. The authors propose a technique for configuring workflow models incrementally, continuously verifying that they are deadlock free. However, their approach does not generalise to checking arbitrary user-defined properties, as we do.

8. CONCLUSION

This paper lays the foundation for scalable modelling and efficient verification of software product lines. We introduced FTS, featured transition systems, a formalism designed to describe the combined behaviour of a whole system

family. While allowing to model very detailed behavioural variations, FTS leverages on treating features as first-class abstractions and supports separation of concerns. A second contribution is a tool-supported model checking approach that allows to verify FTS against LTL properties. Thereby, we are able to verify all the products of a family at once and pinpoint the products that violate properties. An empirical evaluation showed substantial gains over individual product verification. The source code of the Haskell library we implemented is freely available.

9. REFERENCES

- [1] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel. Learning operational requirements from goal models. In *ICSE 31*, pages 265–275, 2009.
- [2] A. Arnold. *Finite transition systems: semantics of communicating systems*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994. Translator-Plaice, John.
- [3] P. Asirelli, M. H. ter Beek, S. Gnesi, and A. Fantechi. Deontic logics for modeling behavioural variability. In *VaMoS'09*, pages 71–76, 2009.
- [4] F. Bachmann, M. Goedicke, J. C. S. do Prado Leite, R. L. Nord, K. Pohl, B. Ramesh, and A. Vilbig. A meta-model for representing variability in product family development. In *Int. Workshop on Product Family Engineering (PPE)*, pages 66–80, 2003.
- [5] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2007.
- [6] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.
- [7] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
- [8] A. Classen. Modelling with FTS: a collection of illustrative examples. Technical report, PReCISE Research Center, University of Namur, Namur, Belgium, September 2009.
- [9] A. Classen, P. Heymans, and P.-Y. Schobbens. What's in a feature: A requirements engineering perspective. In *FASE'08, Held as Part of ETAPS'08*, volume 4961 of *LNCS*, pages 16–30. Springer, 2008.
- [10] A. Classen, P. Heymans, T. T. Tun, and B. Nuseibeh. Towards safer composition. In *ICSE 31, Companion Volume*, pages 227–230. IEEE, 2009.
- [11] P. C. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, August 2001.
- [12] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *GPCE'05*, pages 422–437, 2005.
- [13] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness ocl constraints. In *GPCE '06*, pages 211–220. ACM, 2006.
- [14] C. Ebert and C. Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, 2009.
- [15] A. Fantechi and S. Gnesi. A behavioural model for product families. In *ESEC-FSE'07, Companion*, pages 521–524. ACM, 2007.
- [16] A. Fantechi and S. Gnesi. Formal modeling for product families engineering. In *SPLC 2008*. IEEE CS, 2008.
- [17] D. Fischbein, S. Uchitel, and V. Braberman. A foundation for behavioural conformance in software product line architectures. In *ROSATEA '06, ISSATA 2006 workshop*, pages 39–48. ACM Press, 2006.
- [18] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *CAV 2001*, number 2102 in *LNCS*, pages 53–65, 2001.
- [19] A. Gruler, M. Leucker, and K. Scheidemann. Modeling and model checking software product lines. In *IFIP WG 6.1 FMOODS '08*, pages 113–131. Springer, 2008.
- [20] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, SEI, CMU, November 1990.
- [21] J. Kramer, J. Magee, M. Sloman, and A. Lister. Conic: an integrated approach to distributed computer control systems. *Computers and Digital Techniques, IEE Proceedings E*, 130(1):1–10, 1983.
- [22] K. G. Larsen, U. Nyman, and A. Wasowski. Modal i/o automata for interface and product line theories. In *ESOP*, pages 64–79, 2007.
- [23] M. Mendonca, A. Wasowski, and K. Czarnecki. Sat-based analysis of feature models is easy. In *SPLC'09*, pages 231–240, 2009.
- [24] P. Schnoebelen. The complexity of temporal logic model checking. In *Advances in Modal Logic 4*, pages 393–436, 2002.
- [25] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Feature Diagrams: A Survey and A Formal Semantics. In *RE'06*, pages 139–148. IEEE CS, 2006.
- [26] W. M. P. van der Aalst, M. Dumas, F. Gottschalk, A. H. M. ter Hofstede, M. L. Rosa, and J. Mendling. Correctness-preserving configuration of business process models. In *FASE'08, Held as Part of ETAPS'08*, pages 46–61, 2008.
- [27] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.
- [28] T. Ziadi, L. Hérouët, and J.-M. Jézéquel. Towards a uml profile for software product lines. In *Int. Workshop on Product Family Engineering (PPE)*, pages 129–139, 2003.
- [29] T. Ziadi, L. Hérouët, and J.-M. Jézéquel. Modeling behaviors in product lines. In *Workshop on Requirements Engineering for Product Lines (REPL'02)*, Essen, Germany, Sept. 2002.